Języki i Paradygmaty Programowania

Marcin Benke

19 marca 2018

Monady

```
-- class Applicative m => Monad m where
class Functor m => Monad m where
  return :: a -> m a
  (>>=)  :: m a -> (a -> m b) -> m b
  fail :: String -> m a
```

m a reprezentuje obliczenie, które produkuje wynik typu **a**, potencjalnie z dodatkowymi efektami

Np. **getChar :: IO Char** produkuje znak , wczytując go z wejścia

Efekty

Brak efektu

$$m\ a \simeq a$$

Wyjątki

$$m\ a \simeq a + error$$

Środowisko

$$m\ a \simeq env \to a$$

Stan

$$m\ a \simeq s \to (a, s)$$

Niedeterminizm

$$m\ a \simeq [a]$$

Protokół MonadReader

```
class (Monad m) => MonadReader r m | m -> r where
  ask :: m r
  local :: (r -> r) -> m a -> m a
   -- Defined in Control.Monad.Reader.Class

asks :: (MonadReader r m) => (r -> a) -> m a
```

Protokół MonadReader jest przydatny w sytuacji, gdy mamy grupę funkcji korzystających ze wspólnych danych i chcemy uniknąć jawnego przekazywania ich z funkcji do funkcji

```
-- Defined in Control.Monad.Reader
-- instance MonadReader r ((->) r)
-- (->) r a = r -> a
```

Protokół MonadReader — przykład

```
data Env = Env { var_a :: Int, var_b :: Int }
type R a = Env -> a
-- instance (MonadReader Env) r
example, example2 :: R Int  -- R Int = Env -> Int
example = do
  a <- asks var_a
  b <- asks var_b
  return $ a+b

example2 = liftM2 (+) (asks var_a) (asks var_b)
run :: R a -> a
run r = r Env { var_a = 5, var_b = 10 }

ghci> run example
15
```

MonadReader — local

Metoda **local** pozwala na lokalne zmiany środowiska

```
inc_a :: Env -> Env
inc_a env = env { var_a = var_a env +1 }

ghci> :{
  forM_ [run $ local inc_a example, run example]
    print
:}
16
15
```

`:{` oraz `:}` są komendami ghci pozwalające na wprowadzanie wyrażenia w kilku liniach

MonadReader — przykład

```
-- type Env = Map Name Type
-- instance MonadReader Env TCM

askType :: Name -> TCM Type
askType n = do
  Just t <- asks (Map.lookup n)
  return t

typeOf :: Exp -> TCM Type
typeOf (EVar n) = askType n
typeOf (EInt _) = return TInt
typeOf (ELam n t0 e1) = do
  t1 <- local (Map.insert n t0) $ typeOf e1
  return $ t0 :-> t1
```

Stan

Najbardziej chyba typowym efektem imperatywnym jest możliwość korzystania z globalnego stanu i jego modyfikacji.

Obliczenie ze stanem możemy traktować jako funkcję

```
type Obliczenie wynik = (Stan -> (wynik, Stan))

-- return :: a -> Obliczenie a
return x s = (x,s)

-- (>>=) :: Obliczenie a -> (a -> Obliczenie b)
      -> Obliczenie b
(o >>= k) s = let (x,s') = o s in (k x) s'
```

Typy pozwalają kontrolować gdzie i w jakim zakresie stan może się zmieniać.

Stan

Teraz mozemy zdefiniować np

```
-- Odczyt stanu
czytaj :: Obliczenie Stan
czytaj s = (s,s)

-- Zapis nowego stanu
zapisz :: Stan -> Obliczenie ()
```

```
zapisz s = \_ -> ((),s)

-- Funkcja zmiany stanu jako obliczenie
zmieniacz :: (Stan->Stan) -> Obliczenie ()
zmieniacz t = czytaj >>= \stan -> zapisz $ t stan

zwiekszLicznik :: Obliczenie ()
zwiekszLicznik = zmieniacz (+1)
```

Monada State

Synonimy jako instancje klas są kłopotliwe, zatem używamy newtype:

```
newtype State s a = State{runState :: (s -> (a,s))}

instance Monad (State s) where
    return a        = State $ \s -> (a,s)
    (State x) >>= f = State $
                      \s -> let (v,s') = x s in
                            runState (f v) s'

evalState  :: State  s a -> s -> a
execState :: State s a -> s -> s
```

Protokół MonadState

```
class MonadState m s | m -> s where
    get :: m s
    put :: s -> m ()

instance MonadState (State s) s where
    get   = State $ \s -> (s,s)
    put s = State $ \_ -> ((),s)
```

- **get** odczytuje stan
- **put** modyfikuje stan (zapisuje nowy)

Funkcje pomocnicze gets i modify

Przy programowaniu ze stanem często pojawiają się operacje typu

```
do { ...; s <- get; let x = foo s; ... }
do { ...; s <- get; put (f s); ... }
```

Dlatego **Control.Monad.State** definiuje funkcje

```
gets   :: (MonadState s m) => (s -> a) -> m a
modify :: (MonadState s m) => (s -> s) -> m ()
```

które pozwalają nam napisać zwięźlej i czytelniej

```
do { ...; x <- gets foo; ... }
do { ...; modify f; ... }
```

MonadState — przykład

W procesie typowania potrzebujemy świeżych nazw dla typów

```
-- instance MonadState TCM
-- freshName :: TCM Name

typeOf :: Exp -> TCM Type
typeOf (EInt _) = return TInt
typeOf (ELam n e1) = do
  x <- freshName
  let t0 = TVar x
  t1 <- local (Map.insert n t0) $ typeOf e1
  return $ t0 :-> t1
```

IO — monady zmieniają świat

Monadę IO możemy traktować jako bardzo szczególny przypadek monady **State**:

```
type IO = State RealWorld
```

(dokładniej nie tyle IO jest szczególna, co jej stan — **RealWorld** odpowiada stanowi świata)

Klasa MonadPlus

Klasa **MonadPlus** reprezentuje obliczenia dopuszczające brak wyniku (ew. domyślny wynik) i łaczenie wyników:

```
class Monad m => MonadPlus m where
  mzero :: m a
  mplus :: m a -> m a -> m a

  -- Laws:
  --  mzero >>= f  =  mzero
  --  v >> mzero   =  mzero

guard :: MonadPlus m => Bool -> m ()
guard True  = return ()
guard False = mzero
```

Listy jako monady

Listę możemy traktować jako monadę reprezentującą obliczenia niedeterministyczne (wiele możliwych wyników, relacje)

```
instance Monad [] where
  return a = [a]
  m >>= f  = concatMap f m
  fail s = []
```

Instancja **MonadPlus** dodaje operacje braku wyniku i "niedeterministycznego" wyboru:

```
instance MonadPlus [] where
  mzero = []
  mplus = (++)
```

Listy jako monady

Funkcję **inits** z laboratorium możemy teraz zapisać jako

```
inits :: [a] -> [[a]]
inits [] = return []
inits (x:xs) = return [] `mplus` do
   ys <- inits xs
   (return (x:ys))
```

a nawet

```
inits2 list = return [] `mplus` do
   (x:xs) <- return list -- mzero gdy list == []
   ys <- inits2 xs
   return (x:ys)

inits3 list = []:[(x:ys) |
                  (x:xs) <- [list], ys <- inits3 xs]
```

Listy jako monady

Wycinanki listowe możemy łatwo przetłumaczyć na **do**, np

```
triads :: Int -> [(Int, Int, Int)]
triads n = [(x,y,z) |
             (x,y,z) <- triples n,
             z^2 == x^2 + y^2
           ]

triads2 :: Int -> [(Int, Int, Int)]
triads2 n = do
             (x,y,z) <- triples n
```

```
            guard $ z^2 == x^2 + y^2
            return (x,y,z)

guard :: MonadPlus m => Bool -> m()
guard b = if b then return () else mzero
```

Alternative

Ogólniejszym pojęciem jest `Alternative`:

```
class Applicative f => Alternative f where

    empty :: f a     -- identity of '<|>'
    (<|>) :: f a -> f a -> f a

    -- | One or more.
    some :: f a -> f [a]

    -- | Zero or more.
    many :: f a -> f [a]
```

MonadPlus z Alternative

```
class (Alternative m, Monad m) => MonadPlus m where
   -- | the identity of 'mplus'.  It should also satisfy the equations
   --
   -- > mzero >>= f  =  mzero
   -- > v >> mzero   =  mzero
   --
   mzero :: m a
   mzero = empty

   -- | an associative operation
   mplus :: m a -> m a -> m a
   mplus = (<|>)
```

Rodzaje (kinds)

- Mówimy o konstruktorach typów, że są 0-argumentowe, 1-argumentowe itd
- Można to uogólnić wprowadzając pojęcie *rodzaju* czyli swoistego "typu" dla konstruktorów typów:

$$\kappa = * \mid \kappa_1 \to \kappa_2$$

Typy są rodzaju $*$, konstruktory jednoargumentowe rodzaju $* \to *$ itd:

```
ghci> :kind Int
Int :: *
ghci> :kind Maybe
Maybe :: * -> *
```

KindSignatures

Możemy jawnie podawać rodzaje jeśli użyjemy rozszerzenia **KindSignatures**

```
{-#LANGUAGE KindSignatures #-}
class Functor f => Pointed (f :: * -> *) where
  pure :: a -> f a
```

Łączenie monad (1)

Na laboratorium pisaliśmy różne warianty funkcji

readInts :: String → m [Int]

O analizie syntaktycznej będziemy jeszcze mówić, ale spróbujmy:

```
module StateParser(
  Parser,    -- * -> *
  runParser, -- Parser a -> (String -> m (a,String))
  item       -- Parser Char)

instance Monad Parser where ...
instance MonadPlus Parser where ...
```

Gdzie **m** jest pewną monadą, np **Maybe** (albo **Either**, **List**,...)

Łączenie monad (2)

```
pNat, pDigit :: Parser Int
pDigits :: Parser [Int]
many :: Parser a -> Parser [a]

pNat = fmap (foldl (\x y -> 10*x+y) 0) pDigits
pDigits = many pDigit
pDigit = sat isDigit >>= return . digitToInt

sat :: (Char->Bool) -> Parser Char
sat p = do
  x <- item
  if p x then return x else mzero

> runParser pNat "123 ala"
Just (123," ala")
```

Łączenie monad (3)

Zauważmy, że **Parser** jest podobny do monady **State**, np możemy wyrazić **item** przy pomocy **get** i **put**:

```
instance MonadState [Char] Parser where
...
item :: Parser Char
item = do
  input <- get
  case input of
    [] -> mzero
    (x:xs) -> put xs >> return x
```

Łączenie monad (4)

Możemy na przykład ręcznie połaczyć monady State i Maybe:

```
newtype Parser a = Parser {
   runParser :: [Char] -> Maybe(a,[Char])
   }
mkParser :: ([Char] -> Maybe(a,[Char])) -> Parser a
mkParser = Parser
instance Monad Parser where
  return a = Parser $ \s -> Just (a,s)
  (Parser f) >>= k = Parser $ bind f k where ...
instance Functor Parser where ...
instance MonadPlus Parser where ...
instance MonadState Parser where ...
```

Trzeba się sporo napracować, a co gorsza jeśli będziemy zmienić **Maybe** na **Either e**, to większą część tego kodu będzie można wyrzucić do kosza...

Transformatory monad (1)

Wzorzec łączenia monad wystepuje na tyle czesto, że został usystematyzowany w bibliotece **mtl** (Monad Transformers Library).

W naszym przypadku możemy użyć transformatora **StateT**:

```
module StateTParser(Parser,runParser,item) where
import Control.Monad.State

-- Use the StateT transformer on Maybe
type Parser a = (StateT [Char] Maybe) a

runParser = runStateT
-- instance Monad ... gratis!
-- instance MonadPlus ... gratis!
-- instance MonadState ...gratis!
```

Transformator StateT

```
ghci> :i StateT
newtype StateT s m a = StateT {
   runStateT :: s -> m (a, s)}

ghci> :k StateT
StateT :: * -> (* -> *) -> * -> *
```

Jak widać **StateT** ma dość złożony rodzaj:

- pierwszy argument jest typem stanu i ma rodzaj $*$.
- drugi argument jest transformowanym konstruktorem typu (monadą) i ma rodzaj $* \to *$
- trzeci argument jest typem wyniku obliczenia

Transformatory monad (2)

...dla wiekszej elastyczności możemy też zamiast **Maybe** użyć transformatora **MaybeT**:

```
module StateTParser2(Parser,runParser,item) where
import Control.Monad.State
import MaybeTrans
import Control.Monad.Identity

-- Use the StateT transformer on MaybeT on Identity
type Parser a = StateT [Char] (MaybeT Identity) a

runParser :: Parser a -> [Char] -> Maybe (a,String)
runParser p xs =
  runIdentity $ runMaybeT $ runStateT p xs
```

wtedy możemy łatwo wymienić **Identity** na inną monadę.

Transformatory monad (3)

Teraz możemy łatwo zamiast **Maybe** użyć innego mechanizmu obsługi błędów:

```
...
import Control.Monad.Error

type Parser a = StateT [Char] (ErrorT String Identity) a

runParser :: Parser a -> [Char]
           -> Either String (a,String)
runParser p xs =
```

```
   runIdentity $ runErrorT $ runStateT p xs

testParser :: Show a => Parser a -> String -> String
testParser p xs = show $ runParser p xs
```

Protokół MonadTrans

```
class MonadTrans t where
    lift :: Monad m => m a -> t m a
```

Operacja **lift** pozwala nam "podnieść" obliczenie z monady wewnętrznej do przetransformowanej, np.

```
type Parser a = StateT [Char] (ErrorT String Identity) a

perror :: String -> Parser a
perror = lift . throwError

> runParser (perror "Oczekiwano cyfry") "ala"
Left "Oczekiwano cyfry"
```

Cabal

Common Architecture for Building Applications and Libraries

W tej chwili interesuje nas głównie program cabal, który pozwala instalować biblioteki na swoim koncie, bez uprawnień administratora

```
[ben@students Haskell]$ cabal update
Downloading the latest package list
  from hackage.haskell.org
[ben@students Haskell]$ cabal install GLFW
...kompilacja...
Installing library in
 /home/staff/iinf/ben/.cabal/lib/GLFW-0.4.2/ghc-6.10.4
Registering GLFW-0.4.2...
Reading package info from "dist/installed-pkg-config"
 ... done.
Writing new package config file... done.
```

Wiele bibliotek na `http://hackage.haskell.org/`